中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle 编 译 原 理

第10讲：语法分析(7)

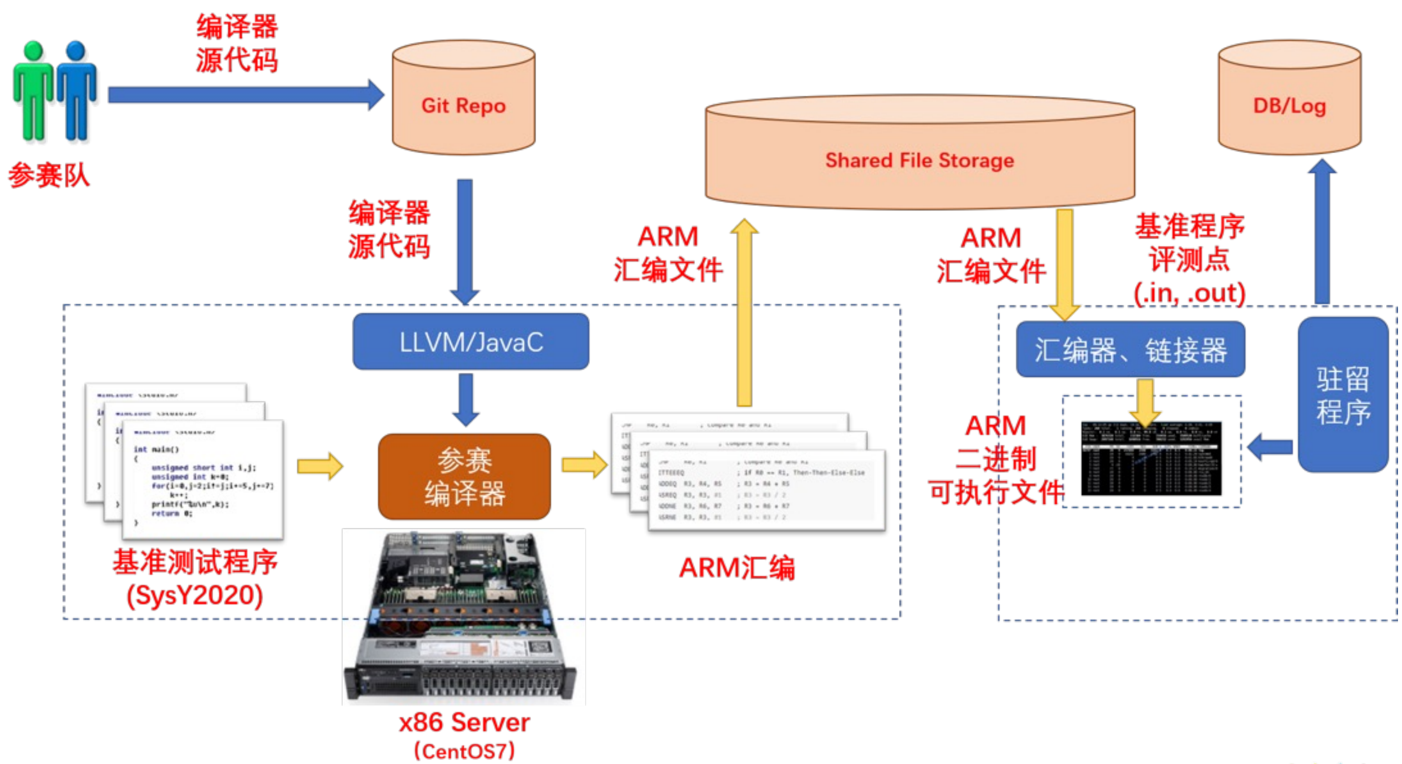
张献伟

xianweiz.github.io

DCS290, 3/28/2023

- 构思并实现一个综合性的编译系统，以展示面向特定目标平台的编译器构造与编译优化的能力
  - 功能测试： 正确编译通过的 SysY2022 基准测试程序
  - 性能测试： 评估每个基准测试在目标硬件平台上的执行时间

报名：3.24 – 5.15
初赛：3.24 – 8.10
决赛：8月中下旬



https://compiler.educg.net/#/index

Review Questions

- Q1: actions in top-down parsing?

  Expand on non-terminal, compare on terminal.

- Q2: how to build the LL(1) parse table?

  Two sets: FIRST, FOLLOW

$S \rightarrow Aa$
$A \rightarrow BD$
$B \rightarrow b$
$B \rightarrow \varepsilon$
$D \rightarrow d$
$D \rightarrow \varepsilon$

- Q3: for the grammar, what is FIRST(A) and FOLLOW(B)?

  FIRST(A) = {b, d, ε}, FO(A) = {a} FOLLOW(B) = {d, a} = FI(D) + FO(A)

- Q4: which one is typically used, LL(0), LL(1), LL(2) …? Why not others?

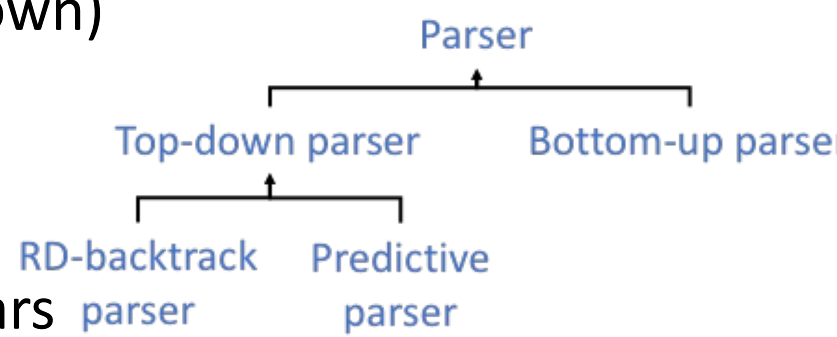
  LL(1). LL(0) is too weak, LL(k) has a too large table.

- Q5: top-down vs. bottom-up parsing?

  Top-down is based on leftmost derivation;
  bottom-up is the reverse of rightmost derivation.

Bottom-up Parsing[自底向上]

- Begins at leaves and works to the top[叶子到根]
  - Bottom-up: **reduce**s[归约] input string to start symbol
  - In the <u>opposite direction</u> from top-down
    - Top-down: expands start symbol to input string
  - In <u>reverse order of rightmost derivation</u> (In effect, builds tree from left to right, just like top-down)



- More powerful than top down
  - Don't need left factored grammars
  - Can handle left recursion
  - Can express a larger set of languages
  - And just as efficient

Bottom-up: Overview

- An important fact:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \to \beta$
  - Then ω is a string of terminals[i.e., 句子]
- Why? $\alpha X\omega \to \alpha\beta\omega$ is a step in a rightmost derivation
- **Idea**: split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)[右侧尚未被解析，i.e., 最右推导中已被完全展开]
  - Left substring has terminals and non-terminals[左侧已有解析，i.e., 最右推导中尚未被完全展开]
- The dividing point is marked by a #
  - The # is not part of the string[仅作为标示]
  - Initially, all input is unexamined $\#x_1x_2 \ldots x_n$[输入尚未有任何解析]

Bottom-up: Shift-Reduce[移入-归约]

- Bottom-up parsing is also known as **Shift-Reduce** parsing
  - Involves two types of operations: shift and reduce
  - Recall: <u>expand</u> and <u>compare</u> operations for top-down parsing

- **Shift**[移入]: move # one place to the right
  - Shifts a terminal to the left string[向左侧移入终结符，推进解析]

    ABC#xyz ⇒ ABCx#yz

- **Reduce**[归约]: apply an inverse production at the right end of the left string[左侧串的右端进行归约]
  - If E → Cx is a production, then

    ABCx#yz ⇒ ABE#yz

The Example

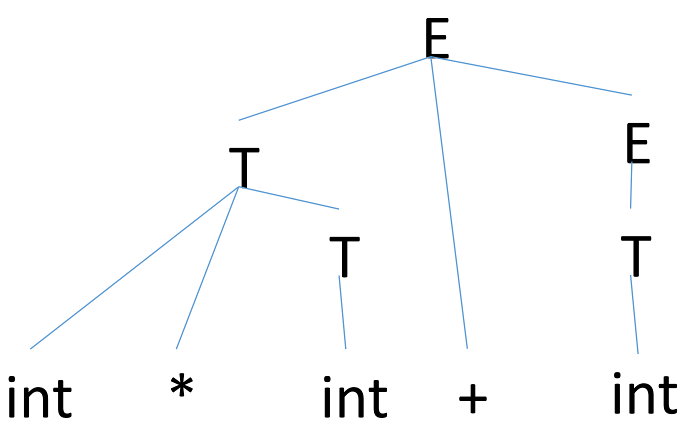
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Stack[栈]

- Left string can be stored into a **stack**
  - Top of the stack is the #[左右串的分界]

- **Shift** pushes a terminal on the stack
- **Reduce** does the following:
  - pops zero or more symbols off of the stack
    - production rhs[pop出了产生式RHS]
  - pushes a non-terminal on the stack
    - production lhs[push进了产生式LHS]
  - just reverts production (LHS ← RHS)[产生式逆向使用]

Step
#int * int + int
int# * int + int
int * #int + int
int * int # + int
int * T # + int
T # + int
T + # int
T + int #
T + T #
T + E #
E #

Key Issue[一个关键问题]

- How to decide when to shift or reduce?
  - Example grammar:

    E → T+E | T

    T → int*T | int | (E)

#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
… … …	✓

#int * int + int	Shift
int# * int + int	Reduce T → int
T # * int + int	Shift
… … …	✗

  - Consider the step int # * int + int
  - We could reduce by T → int giving T#*int + int
    - **A fatal mistake**: no way to reduce to the start symbol E

- **Intuition**: want to reduce only if the result can still be reduced to the start symbol[必须在对的方向上] !?

The Example

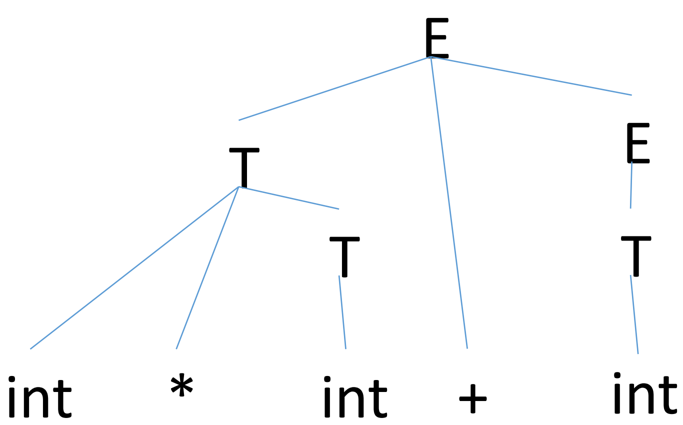
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Why not just E → T?
(int * T => int * E)

Handle[句柄]

- A **handle** of a sentential form is a substring α such that:
  - *α* matches the RHS of a production *A -> α* ; and[能匹配上规则]
  - replacing *α* by the LHS *A* represents a step in the reverse of a rightmost derivation of S[且能推进解析]
- Definition: let αβω be a sentential form where:
  - α, β is a string of terminals and non-terminals (yet to be derived)
  - ω is a string of terminals (already derived)
  - Then β is a **handle** of αβw if: $S \Rightarrow^*_{rm} \alpha X \omega \Rightarrow \alpha\beta\omega$ by a rightmost/rm derivation (apply rule $X \rightarrow \beta$)
- ☞ We only want to reduce at handles, and there is exactly one handle per sentential form
  - But where to find it?

Some Concepts[一些概念]

- A **right-sentential form**[最右句型] is a sentential form that occurs in the rightmost derivation of some sentence
- A **phrase**[短语] is a subsequence of a sentential form that is eventually "reduced" to a single non-terminal
  - β is a phrase of the right sentential form γ iff $S \Rightarrow^*_{rm} \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - a string consisting of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree[一个句型的语法树中任一子树叶结点所组成的符号串都是该句型的短语]
- A **simple phrase**[直接短语] is a phrase that is reduced in a single step
  - β is a simple phrase of the right sentential form γ iff $S \Rightarrow^*_{rm} \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
- The **handle** is the leftmost simple phrase[最左直接短语]

Handle: Example

- Grammar

  E → T+E | T

  T → int*T | int | (E)

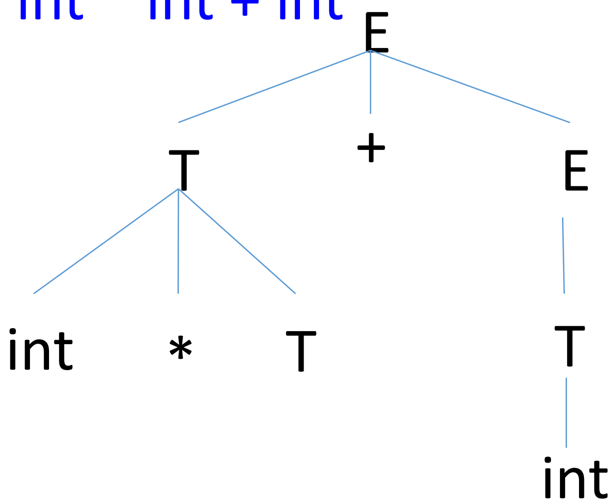
- String

  int * int + int

Step	**Operation**
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * **int** # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + **int** #	Reduce T → int
T + **T** #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Handle: Example

- Grammar
  - E → T+E | T
  - T → int*T | int | (E)
- String
  - int * int + int



Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * **int** # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + **int** #	Reduce T → int
T + **T** #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Handle: Example

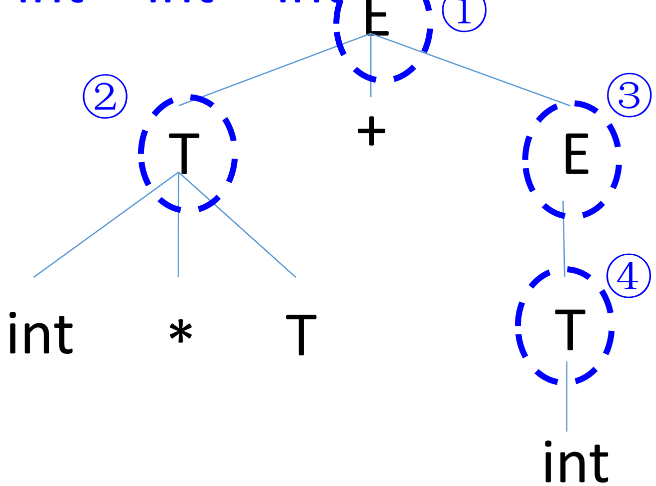
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

Handle: Example

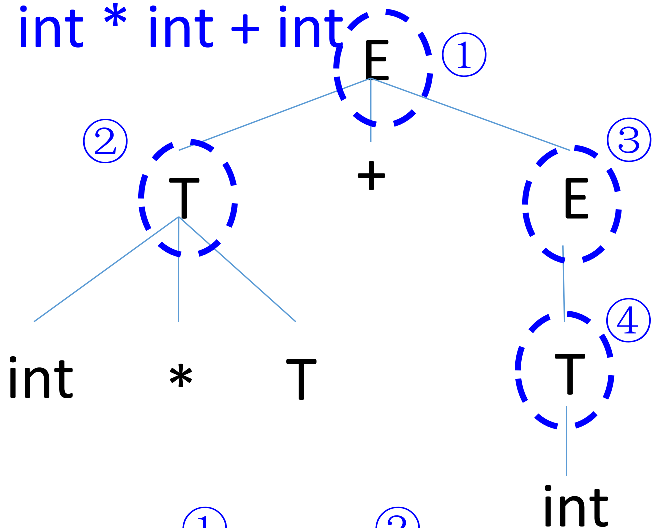
- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int



①　②　③④

Phrase: int * T + int, int * T, int

Simple phrase: int * T, int

Handle: int * T

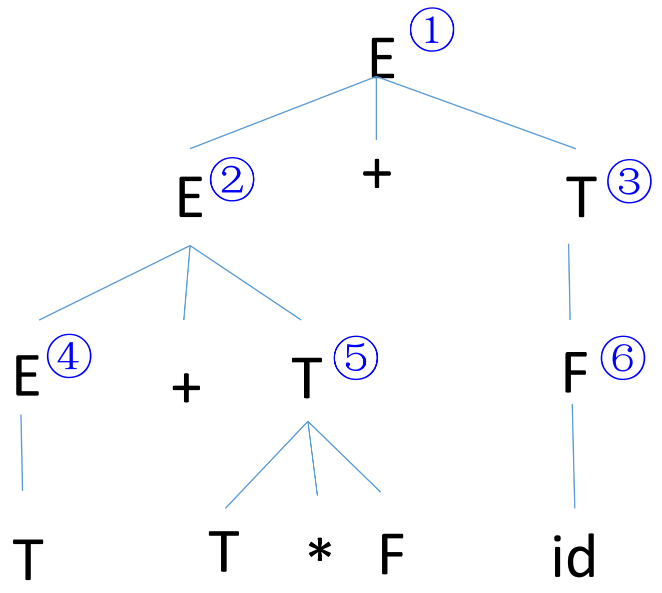
Step	Operation
#int * int + int	Shift
int# * int + int	Shift
int * #int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift
T + # int	Shift
T + int #	Reduce T → int
T + T #	Reduce E → T
T + E #	Reduce E → T+E
E #	

One More Example

- Grammar
  - E → E+T|T
  - T → T*F | F
  - F → (E) | id
- Sentential form:
  - T + T*F + id
- Phrase:
  - T+T*F+id①
  - T+T*F②
  - T④
  - T*F⑤
  - id③⑥

Handle

Simple phrase



Handle Always Occurs at Stack Top

- Why can't a handle occur on right side of #?[不在栈外]
  - It can
  - But handle will eventually be shifted in, placing it at top of stack
  - In int * #int + int ⇒ int * int # + int, int is eventually shifted to the top (i.e., left to #)
- Why can't a handle occur on left side of #, i.e., in middle of the stack?[不在栈'中']
  - Can int * int + # int occur? NOPE.
  - Means parser shifted when it could have reduced when the handle was on top[本应归约却移入]
  - If eagerly reduces when handle is at top of stack, never occurs
- ☞ Makes life easier for parser (need only access top of stack)

int * #int + int	Shift
int * **int** # + int	Reduce T → int
int * T # + int	Reduce T → int*T

Viable Prefix[活前缀]

- In shift-reduce parsing, the stack contents are always a **viable prefix**[活前缀/可动前缀]
  - A prefix of some right-sentential form that ends no further right than the end of the handle of that right-sentential form
    - A viable prefix has a handle at its rightmost end
  - Stack content is always a viable prefix, guaranteeing the shift / reduce is on the right track[活前缀说明移进归约是正确的]
- 定义：一个可行前缀是一个最右句型的前缀，并且它没有越过该最右句型的最右句柄的右端
  - 举例：S => bBa => bbAa，这里句柄是 bA，因此可行前缀包括 bA 的所有前缀（包括 b, bb, bbA），但不能是 bbAa（因为越过了句柄）

... => T + int => int * T + int
Handle: int * T
Viable prefix: int, int *, int * T

int * #int + int	Shift
int * int # + int	Reduce T → int
int * T # + int	Reduce T → int*T
T # + int	Shift

Ambiguous Grammars[二义文法]

- Conflicts arise with ambiguous grammars
  - Bottom up parsing predicts action w/ lookahead (just like LL)
  - If there are multiple correct actions, parse table will have conflicts
- Example:
  - Consider the ambiguous grammar E → E * E | E + E | (E) | int

Sentential form	Actions
int * int + int	shift
…	…
E * E # + int	**reduce E → E * E**
E # + int	shift
E + # int	shift
E + int #	reduce E → int
E + E #	reduce E → E + E
E #	
	First x then +

Sentential form	Actions
int * int + int	shift
…	…
E * E # + int	**shift**
E * E + # int	shift
E * E + int #	reduce E → int
E * E + E #	reduce E → E + E
E * E #	reduce E → E * E
E #	
	First + then x

Ambiguous Grammars (cont.)

- In the red step shown, can either shift + or reduce by E → E * E
  - Both okay since precedence of + and * not specified in grammar
  - Same problem with associativity of + and *
- As usual, remove conflicts due to ambiguity ...
  - 1. rewrite grammar/parser to encode precedence and associativity[指定优先级和结合性]
    - Rewriting grammar results in more convoluted grammars
    - Parser tools have other means to encode precedence and association
  - 2. get rid of remaining ambiguity (e.g. if-then-else)
    - No choice but to modify grammar
- Is ambiguity the only source of conflicts?
  - Limitations in lookahead-based prediction can cause conflicts
  - But these cases are very rare

Properties of Bottom-up Parsing[属性]

- Handles **always** appear at the **top of the stack**
  - Never in middle of stack
  - Justifies use of stack in shift – reduce parsing
- Results in an easily generalized **shift – reduce** strategy
  - If there is no handle at the top of the stack, shift[无句柄，移入]
  - If there is a handle, reduce to the non-terminal[有句柄，归约]
  - Easy to automate the synthesis of the parser using a table
- Can have conflicts[冲突可能发生]
  - If it is legal to either shift or reduce then there is a <u>shift-reduce conflict</u>[移入-归约冲突]
  - If there are two legal reductions, then there is a <u>reduce-reduce conflict</u>[归约-归约冲突]
  - Most often occur because of ambiguous grammars
    - In rare cases, because of non-ambiguous grammars not amenable to parser

Types of Bottom-Up Parsers[类型]

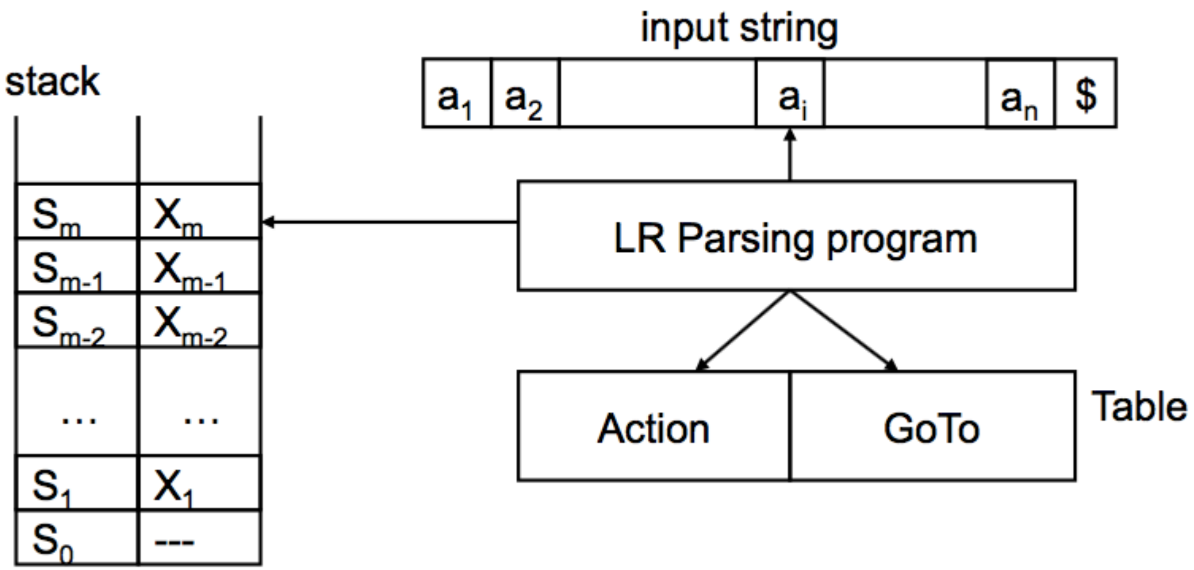
- Types of bottom up parsers
  - Simple precedence parsers[简单优先解析器]
  - Operator precedence parsers[运算符优先解析器]
  - Recursive ascent parsers[递归提升解析器]
  - LR family parsers[LR类解析器]
  - ...

- In this course, we will only discuss **LR family parsers**
  - Efficient, table-driven shift-reduce parsers
  - Most automated tools for bottom-up parsing generate LR family
  - Categories: LR(0), LR(1), SLR, LALR, ...

LR(k) Parser

- **LR(k)**: member of LR family of parsers
  - L: scan input from left to right
  - R: construct a rightmost derivation in reverse
  - k: number of input symbols of lookahead to make decisions
    - k = 0 or 1 are of particular interests, is assumed to be 1 when omitted
- Comparison with LL(k) parser[对比]
  - Efficient as LL(k)
    - Linear in time and space to length of input (same as LL(k))
  - Convenient as LL(k)
    - Can generate automatically from grammar – YACC, Bison
  - More complex than LL(k)
    - Harder to debug parser when grammar causes conflicting predictions
  - More powerful than LL(k)
    - Handles more grammars: no left recursion removal, left factoring unneeded
    - Handles more (and most practical) languages: LL(1) $\subset$ LR(1)

LR Parser



- The stack holds a sequence of states, $s_0s_1...s_m$ (s_m is the top)
  - States are to track where we are in a parse
  - Each grammar symbol X_i is associated with a state s_i
- Contents of stack + input ($X_1X_2...X_ma_i...a_n$) is a right sentential form
  - If the input string is a member of the language
- Uses $[S_m, a_i]$ to index into parsing table to determine action

Parse Table[分析表]

- LR parsers use two tables: **action table** and **goto table**
  - The two tables are usually combined
  - Action table specifies entries for terminals
  - Goto table specifies entries for non-terminals

- Action table[动作表]
  - Action[*s*, *a*] tells the parser what to do when the state on top of the stack is *s* and terminal *a* is the next input token
  - Possible actions: **shift**, **reduce**, **accept**, **error**
- Goto table[跳转表]
  - Goto[*s*, *X*] indicates the new state to place on top of the stack after a reduction of the non-terminal *X* while state *s* is on top of the stack

Possible Actions[可能动作]

- **Shift**
  - Transfer the next input symbol onto the top of the stack
- **Reduce**
  - If there's a rule $A \rightarrow w$, and if the contents of stack are qw for some q (q may be empty), then we can reduce the stack to qA
- **Accept**
  - The special case of reduce: reducing the entire contents of stack to the start symbol with no remaining input[完全归约到开始符号]
  - Last step in a successful parse: have recognized input as a valid sentence[输入串被识别为符合语法]
- **Error**
  - Cannot reduce, and shifting would create a sequence on the stack that cannot eventually be reduced to the start symbol